A Generative Approach For Development Of Test Coverage Analyzers

by

UTPAL BHATTACHARYYA

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUI

January, 1997

A Generative Approach For Development Of Test Coverage Analyzers

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by
Utpal Bhattacharyya

to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur
January, 1997

CSE-1997-M-BHA-GEN

Certificate

Certified that the work contained in the thesis entitled "**A Generative Approach For Development Of Test Coverage Analyzers**", by Mr.*Utpal Bhattacharyya*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

[illegible]

(Dr. Sanjeev K. Agarwal)
Associate Professor,
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

January, 1997

Acknowledgments

First of all, I would like to express my sincere gratitude to Dr. Sanjeev K. Agarwal, for his invaluable guidance and constant encouragement throughout the work. I would also like to thank him for giving me the opportunity to undertake this challenging topic, as my M. Tech. thesis. It was his wonderful association, that enabled me to achieve the objectives of this work.

I am also grateful to Dr. Pankaj Jalote, for introducing me to the exciting field of Software Engineering, which helped me to build interest in Software Engineering tools.

I do appreciate the persons involved in developing Public Domain Softwares, and making them available in the Internet. Without the help of Internet and public domain softwares, it would have not been possible to complete this work in such a short time.

I would also like to thank the persons at the ERNET Lab, IIT Kanpur, for providing Internet access to the students. Thanks to Brahmaji Rao, S.M. Pandit, and the other Lab staff of the Department, for their kind corporation during the thesis work.

Special thanks to Jayesh, Deepak, Samir, Warsi, Sameer Shah, Milind, Kommu and all my friends of MTech'95 batch, for their constant assistance and also for giving memorable company during my stay at IIT Kanpur.

Last, but not the least, I do acknowledge the constant support and encouragement from my parents, brother, sister and brother-in-law. It was their emotional support, that gave me the spirit to carry on my study at IIT Kanpur.

Utpal Bhattacharyya

Abstract

Software Test Coverage Analyzers are very much useful in software testing process. It can speed-up the software testing process. But, they are language dependent tools; same Test Coverage Analyzers cannot be used for testing programs written in different languages. However, in today's era, when people are too much concerned about productivity and quality, one cannot do away with Test Coverage Analyzers in a testing process, specially when it comes to testing big and complicated projects.

Development of Test Coverage Analyzers is a labour intensive and time-consuming task. However, it has been observed that basic development process of a Test Coverage Analyzer is same, irrespective of the language for which it is developed. This indicates the potential for automatic development of Test Coverage Analyzers. This thesis proposes an approach for generating Test Coverage Analyzers. The proposed approach has been used to design and implement a tool, which generates Test Coverage Analyzer for a language. This tool reads two specification files, and generates the Test Coverage Analyzer for the language. The specification files contain

- *Probe specification, giving details of the probes and the places for them.*
- *Grammar specification(YACC) for the language, for which the Test Coverage Analyzer is to be developed.*

The tool provides high development productivity, and also assures the quality of the generated Test Coverage Analyzers. Application experiences of the tool have shown that, a 150 lines(approximately) specification is enough for generating a Test Coverage Analyzer; and the productivity gain is as much as 25-35 times of the hand coded development.

Contents

List of Figures

Chapter 1

Introduction

1.1 Need of Software Testing

The development of software systems involves a series of activities which are prone to human errors. Errors may occur at the requirement phase, design phase as well as in coding phase [Jal91]. Detection of errors in a Software requires lot of effort, as it is not a physical object [Bei90]. Studies have shown that, software testing consumes nearly half of the effort required to produce a working software system [Jal91, Bei90]. Due to these reasons, software development is accompanied by quality assurance activity, and Software Testing is a critical element of software quality assurance.

Software Testing can be defined as a process of executing a program with the intent of finding errors [Mye79]. A successful test is one which uncovers an as yet undiscovered error. Hence, testing should be done with well-planned test cases, which ensures execution of all possible basic paths in the program. This is a time consuming task; software testing tools, that can reduce the testing time without reducing the thoroughness are very much valuable in this context.

There are two basic approaches for Software testing, namely, *structural testing or white box testing* and *functional testing or black box testing*. Structural testing compares the test program behavior against the apparent intention of the source code, taking into account the possible pitfalls in structure and logic. Structural testing is also known as path testing. On the other hand, functional testing examines what the program accomplishes, without any regard to how it works internally. That is, functional testing is concerned with examining the behavior of the program with respect to the requirement specification.

1.2 Test Coverage Analyzers

In a software testing process, an important decision to be taken is when to stop testing. The testing process can be said to be over if the test cases meet the desired level of coverage of the different constructs. That is exactly what a Software Test Coverage Analyzer does. A Test Coverage Analyzer automates the process of [Cor96]:

- finding the areas of a program, that are not covered by test cases
- determining quantitative measure of test coverage, and
- creating additional test cases to increase coverage.

Another aspect of Test Coverage Analyzer is to eliminate redundant test cases. Test Coverage Analyzers are used for assuring the quality of the test cases, which assists in assuring the quality of the actual product.

The testing technique automated by a Test Coverage Analyzer is known as Test Coverage Analysis. Some of the fundamental assumptions behind this technique are

- faults relate to control flow and can be exposed by varying the control flow [Bei9
- A successful test run implies program correctness [Mor90]. But it does not mean that program is free of bugs; the quality of the test case should also be measured.
- The test program should not have any unreachable codes.

Test Coverage Analysis is a structural testing technique. Basically, a Test Coverage Analyzer takes a source program as input, and inserts software probes into the source code. Using these software probes, it monitors the test run of the program and determines the coverage measures. The ideal places in a program, where the probes to be put are the places, where transfer of control takes place (e.g branch statement, procedure call, relational expression, task entry point etc.), so that traversal of any basic path can captured. However, optimal probe insertion techniques are available in the literature [RKC75, Pro82, Aga94].

1.3 Some Of The Basic Coverage Measures

There are a large variety of Coverage measures used by different Test Coverage Analyzers. Detailed discussion of all these measures is out of the scope of this report. Brief ideas about some of the basic coverage measures are given below.

Statement Coverage : This measure - also known as line coverage, basic block coverage, or segment coverage - reports whether each executable statement is executed or not, for a given test case. The main advantage of this method is that it can be directly applied to the object code and does not require the processing of the source code. Performance profilers commonly implement this measure. This coverage measure is insensitive to the some control statements conditions; it cannot check whether all the branch options has been covered or not [Cor96].

Branch Coverage : This measure reports whether all the branch possibilities are explored by the test case or not. The entire boolean condition in a branch statement is treated as a single true-and-false predicate and coverage reporting is done on the basis of that. This category also includes coverage measures for switch statement's cases and exception handlers. Branch Coverage is also known as decision coverage, basic path coverage or all-edge coverage. This type of coverage does not consider the branches within the boolean expression, that may occur due to the logical operators [Cor96]. For example consider the following code segment

```
if ( condition1 && (condition2 || function1()))
        statement1;
else
        statement2;
```

In this example, the test condition is true when condition1 and condition2 are true, and false when condition1 is false. So, in such situations, there is no need to make a call to function1(). The branch coverage measure may consider the control structure completely exercised, without a call to function1().

condition coverage : This measure reports the true or false outcome of every boolean subexpression separated by logical operators. It reports each subexpressions separately. The advantage of this measure is the thoroughness in testing [Cor96].

Path Coverage : This measure reports whether all the possible paths in each function or procedure has been executed or not. A path is a unique sequence of branches from the entry point to the exit point of a module(i.e. function or procedure). This measure achieves the maximum thoroughness in testing; but the number of paths in a program is exponentially related to the number of branches and hence requires a large number of software probes to monitor all the paths. For example, a function, containing just 10 if statements, has 1024 paths to test. Adding just one more is statement, it goes up to 2048. Another disadvantage is that many paths are impossible to exercise due to relationship of data. For example, consider the following code segment.

```
if ( success )
        statement1;
statement2;
if(success)
        statement3;
```

Path coverage measure will show that there are four possible paths. But, ir reality, there are only two paths($success == true$ and $success == false$).

Function Coverage : This measure reports whether each procedure and function defined in the program has been invoked or not.

Call coverage : This measure reports whether each procedure/function call in the program has been executed or not. The assumption behind this measure is that, faults may occur in interfaces between modules. This measure is different from the previous one in the sense that function coverage deals with measuring the coverage of the functions/procedures defined in the program, not the procedure/function calls.

Loop Coverage : This measure reports about how many times the body of a loop has been executed for a a test case.

Data flow coverage : This is a variation of path coverage. This measure deals with coverage of sub-paths from a variable assignments to its subsequent references. This measure has direct relevance to the way a program handles the data; but it does not consider the branch coverage. Further, this measure is very complex to implement.

Race Coverage : This measure reports whether multiple threads/task is executing the same code simultaneously. This measure helps in detecting the faults in multi-threaded programs.

Relational Operator Coverage : This measure reports whether boundary situations occur with the relational operators. That is, it reports the exact condition which causes a particular branch evaluation. The assumption behind this measure is that boundary test cases find off-by-one errors and incorrect uses of relational operators such as < instead of <=. For example consider the following C++ code segment.

```
if (a < b)
        statement;
```

Relational operator coverage reports whether the boundary condition a==b occurs or not.

Weak mutation Coverage : This measure is a generalized form of Relational Operator Coverage. It tries to expose the use of wrong operators and operands. It works by substituting(mutating) the expressions in the program with the alternative operators(e.g. "-" substituting "+") and also with alternative variables for reporting the coverage of the conditions [How82].

Table Coverage : This measure reports whether each entry in a particular array has been referenced or not. This is useful for testing programs which are controlled by finite state machines, such as parser.

1.4 Setting Up Coverage Goals

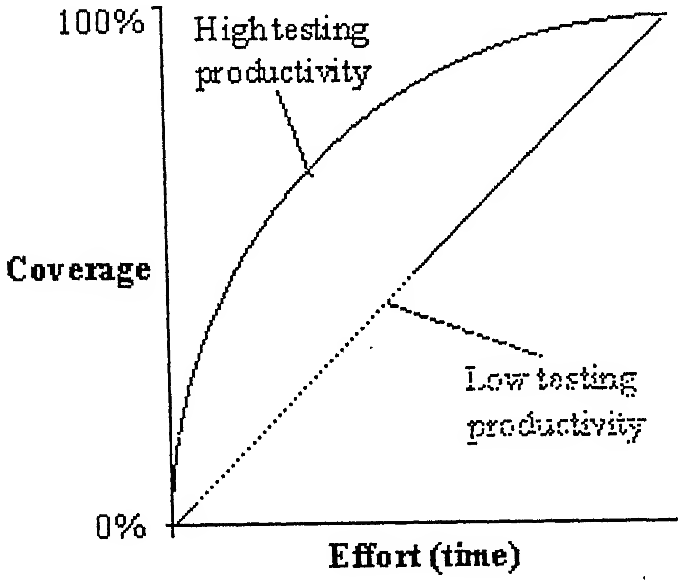

Figure 1.1: Coverage rate(Courtesy[Cor96])

It is important to set up the coverage goal for each product, in order to assure the quality of the product. Each project must meet some minimum percentage coverage criteria for release, so as to prevent post-release failures [Cor96]. Of course, this percentage is dictated by the testing resources and the nature of the product. For example, a safety-critical software must have higher coverage goal.

Apart from the final coverage goal, it is also necessary to set up intermediate coverage goals as it helps in increasing the testing productivity. Testing productivity increases, if and only if most of the failures can be found with the least effort(includes, time required to create test cases, add in the test suit, and run them). Test coverage analyzers do help in achieving this. They provide the highest probability of finding bugs in least time. Figures 1.1 and 1.2 shows the Coverage rate and failure discovery rates for low and high productivity testing respectively.

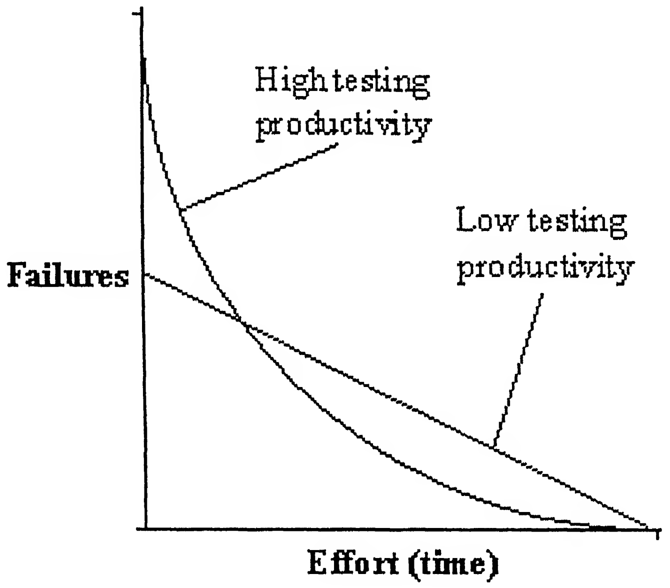

Figure 1.2: Failure Discovery rate(Courtcsy[Cor96])

One way to achieve high testing productivity is to first attain some coverage throughout the entire source program before going for high coverage for any particular area in the program. The hypothesis is to look for failures which can be easily found by minimal testing. Coverage goals for these intermediate testing session should be set up to achieve effective and high-productivity testing.

1.5 Coverage Measures Incorporated In This Work

In this thesis work, we have designed a Test Coverage Analyzer for Ada'95, and a tool for automatic generation of Test Coverage Analyzers. Present implementation incorporates the following coverage measures.

- Function Coverage
- branch coverage
- loop coverage
- Call Coverage
- block-wise coverage

The detailed discussion of this work is carried out in the following chapters.

1.6 Organization Of The Thesis

This thesis has been organized as follows. The next chapter gives a brief idea about the ongoing works in the development of Test Coverage Analyzers. In this chapter, a brief overview about the features of some of the the well-known Test Coverage Analyzers are given. We have also looked into the works that have been already done in IIT Kanpur. Finally, a few words about our thesis work, its philosophy, and the motivation behind this work, are outlined in this chapter.

Chapter 3, describes a design approach for Test Coverage Analyzers. The step-by-step approach for the design, with an intention to find out the potential for automating these step, is outlined in this chapter. The design discussed in this chapter has been used to design and implement a Test Coverage Analyzer for Ada'95. A brief discussion about the implementation issues is also carried out in this chapter. Finally, the scope for the automatic generation has been pointed out here.

The fourth chapter enumerates the issues in design and the implementation of a tool for Automatic generation of Test Coverage Analyzers. A comprehensive design for achieving this is also discussed here.

Chapter 5, speaks about the application experiences of both the tools implemented under this thesis work. The results of the performance evaluation of the generated Test Coverage Analyzers for the languages C, and Ada'95, in comparison to hand-coded Test Coverage Analyzers, are also outlined in this chapter.

The complete user's manual for GCTA(Generic Test Coverage Analyzer, the tool used for generating Test Coverage Analyzer), is given in Appendix A. Appendix B gives the User's manual for ADACOV(Test coverage analyzer for ADA'95), and also speaks about how to interpret output of the coverage tool.

Chapter 2

Background

This chapter gives a brief overview of some of the existing commercial and non-commercial testing tools, that support Test Coverage Analysis. A brief introduction to our work, and the philosophy behind it, are also outlined in this chapter.

2.1 A Survey Of Some Well-known Test Coverage Analyzers

There are a number of Test Coverage Analyzers in the commercial/non-commercial software product domain; and it is not possible to describe all of them. Some of the well-known Test Coverage Analyzers, and their features are enumerated below:

AdaTEST : This is a widely used Testing tool, from Information Processing Limited[1](IP for Ada programs. This is a complete software testing environment with the capabilities of dynamic testing, coverage analysis, and static analysis. As dynamic Testing, AdaTEST supports functional testing, structural testing and the timing analyses. AdaTEST provides the facilities for coverage analysis of single units, package, tasks, and the entire Ada System. It incorporates statement, branch and condition coverage. It also provides code audit facilities for assessing compliance with coding standard and measuring code complexity, as

[1]http://www.iplbath.com/

well as calculating many of the published matrices(e.g. McCabe, Halstead); it can also be configured to use non-standard(i.e. user-defined) matrices in its analysis. The testing results are reported in an easy to read form. Currently, work is going on to incorporate testing capabilities of Ada'95 programs, and the name of the tool has been provisionally kept as AdaTEST'95.

Cantata : This is a Software Testing tool Developed and marketed by IPL, and provides a complete Software Testing environment for C and C++ programs. The capabilities are exactly same as AdaTest, and supports dynamic testing, coverage analysis, and static analysis for C and C++ programs.

C-Cover : This is a Software Test Coverage Analyzer, developed and marketed by Bullseye Testing Technology[2], for C and C++ Programs. The tool incorporates the capabilities for branch, function and condition coverage. Unlike Cantata, It does not support static analysis and complexity measures.

GCT : GCT(stands for Generic Coverage Tool) is a free-ware coverage tool that measures how thoroughly tests exercise C programs. It supports the following coverage measures

- branch coverage
- multiple-condition coverage.
- loop coverage
- relational operator coverage
- function coverage
- call coverage
- race coverage
- weak mutation coverage

GCT has been widely used in production environments, including testing the UNIX kernel. The tool takes C source code, adds instrumentation, then uses

[2]http://www.bullseye.com/

the system C compiler(or a compiler of user's choice) to produce an object file. Alternately, the instrumented source can be captured and moved to another machine for compilation. GCT is designed to work with existing makefiles(or other system build mechanisms). Most makefiles will require no changes. For the sake of efficiency, coverage information is stored in an internal log. The log is written on exit(or, in the case of an unending program like the UNIX kernel, extracted from the running program).

GCT is one of the most versatile Test Coverage Analyzers and is a public domain software developed under GNU general public license agreement. The details of GCT can be obtained from GCT documentation [Mar96].

2.2 Work Done At IIT Kanpur

In the Department of Computer Science and Engineering, at IIT Kanpur, some work has been done in the area of Software Test Coverage Analysis. A Test Coverage Analyzer for C, named CCOV [Sha91, Pal93], and test data generator [Pal93] have been designed and subsequently implemented. Works on several static analysis tools for complexity measurement, style tool etc. have also been carried out here.

2.3 About This Thesis Work

After reviewing the current works in the domain of Test Coverage Analyzers, it has been found that all these works are more or less similar, with some variation regarding the coverage measures. At the same time, we noticed that the development of Test Coverage Analyzers is a labour-intensive and time-consuming task. For example, the total size of the source code for GCT, is about 6MB. As the development approach for all these tools are more or less similar, there is a possibility of a generative approach for developing Test Coverage Analyzers. Keeping this idea in mind, we looked through several journals and Internet to find whether any work of this kind has been going on or not. And we found that, no work is going on regarding automatic generation of Test Coverage Analyzers. This is the basic motivating force for taking up this challenging topic under this work.

Significant work has been going on in the compiler domain, regarding its automatic generation. Compiler generators like GAG [KHZ82], COCO [Mos90] etc. are a few of the examples. These tools read some sort of specification file, and generate compilers, or one or more components of a compiler. Our objective is to develop similar kind of tool for Test Coverage Analyzers, which will generate Test Coverage Analyzer for a language after reading some sort of specification. The philosophy behind our thesis work is to ease the work of developer for developing a Test Coverage Analyzer. Further, programming is not reliable, and the reliability of any software decreases with the increase in code size; and Software Test Coverage Analyzers involve lot of programming. One of the objectives of this work is to minimize programming from the development process. Instead of that, if we can provide some well-tested software, which can generate test coverage analyzer with minimal information from user side, it would provide an efficient and easy way for developing Test Coverage Analyzers.

A generative approach has the highest potential for productivity [PD93], and at the same time, it reduces the cost of the software(usually, the commercial testing tools are very costly). Apart from productivity gain, the generated software also assures quality.

This thesis work is centered around the automatic generation of Test Coverage Analyzer. We designed and subsequently implemented a generator for accomplishing this task. The generator takes a specification and the grammar rules as input, and generates the Test Coverage Analyzer for the language specified by the grammar.

The detail discussion about our work has been given in the following chapters.

Chapter 3

A Design Approach For Test Coverage Analyzers

Methodologies adopted for designing Test Coverage Analyzers are more or less similar irrespective of the language for which it is designed. Our intention is to find out the steps that are carried out manually and issues regarding coverage monitoring, maintenance and reporting. With this motivation in mind, we describe an approach for designing a Test Coverage Analyzer in this chapter. The proposed approach has been used to design and subsequent implementation of a Test Coverage Analyzer for Ada95 [Whe96].

3.1 A Design Scheme

3.1.1 Various Phases in The Design Process

The basic task of a Test Coverage Analyzer(TCA) is to find the places, where the probes are to be put, and instrument the source program, with an intention of incorporating certain coverage measures. To accomplish this task, it needs to parse the source program. A comprehensive description of different parsing techniques are given in the "Dragon Book" [ASU86]. After identifying the place, TCA instruments the source program, by inserting some software monitors. A block diagram showing various phases for development of Test Coverage Analyzer, and the overall architecture is shown in Figure 3.1.

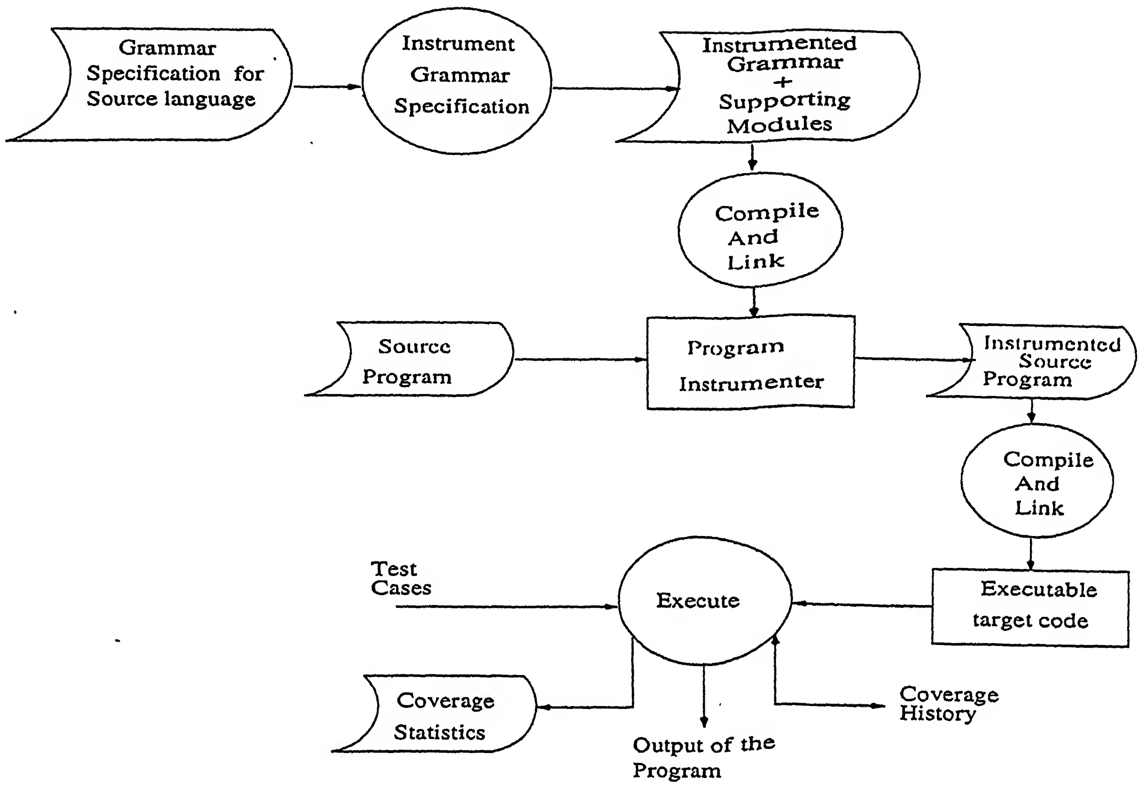

Figure 3.1: Functional diagram of Test Coverage Analyzer

The grammar specification file contains yacc specification for the grammar of the source language - in this case Ada95.

The process "Instrument Grammar file" is the most important and crucial one in the whole development cycle. Basically it involves three main activities, and are normally carried out manually. They are :

- *Identifying places for probe actions*, in the grammar(YACC/BISON specification) file.
- *Inserting probe actions*, and
- *Writing supporting modules* and data structures

Identifying Places for Probe Actions

This activity is dictated by the coverage measures to be incorporated by the Test Coverage Analyzers. Our design of TCA for Ada95, takes into account of Branch coverage, Call coverage, Function Coverage and Loop coverage. Thus, the places for the probes are at the points where transfer of control takes place(for branch coverage and loop coverage), at the entry point of a procedure and function definition(for function coverage) and just after procedure and function call statement(for Call coverage). The YACC production rules for these places has to be identified, in order to put probe actions.

Type of the block	Begin linenumber	End linenumber	tabs	depth

Figure 3.2: Structure of Block Table

Inserting Probe Actions

This activity is concerned with inserting probe actions in the selected production rules. The nature of this action may vary depending upon the design criteria. Our design adopts a two-pass strategy for inserting probes into the source program. In the first pass, a table is constructed to contain the information about the various blocks and statements at which the software monitors are to be put. This table is named as "block table", whose structure is shown in figure 3.2. This table stores the information about the statement type(e.g. if_statement while_statement etc.), whether begining or ending of a block, line number, tab setting(for beautifying the instrumented source program), and the depth of the block(nesting level). The depth information is used latter for showing the skeleton block structure of the test program, which can be used for verifying the coverage results. The probe actions to be inserted in the selected production rules are concerned with constructing this table, and storing the locations of the probes to be inserted in the source program.

The software probes are inserted in the instrumented source file during the second scanning(i.e. pass 2) of the source file. How this task is accomplished, is discussed in the next section.

▪ Writing Supporting Modules

This activity involves, writing supporting routines for generating instrumented program. These routines read source program and *block table*, and inserts probes in the test program. Writing modules for generating the supporting routines for the probes in the test program are also included in this activity. These routines should be in the language used by the test program.

At the end of the phase "Instrument Grammar file", we have the instrumented grammar file, and supporting modules, which, if compiled, gives a module called "Program Instrumenter". "Program instrumenter" generates the instrumented source program. In this thesis, this module is synonymously used with Test Coverage Analyzer.

3.1.2 Instrumenting Source Program

Software monitors are inserted into the source program by the software module called "Program Instrumenter". Figure 3.3 shows the functional architecture of this module. Basically, this module consists of three components. They are,

- Source Program Reader
- Probe locator, and
- Instrumenter

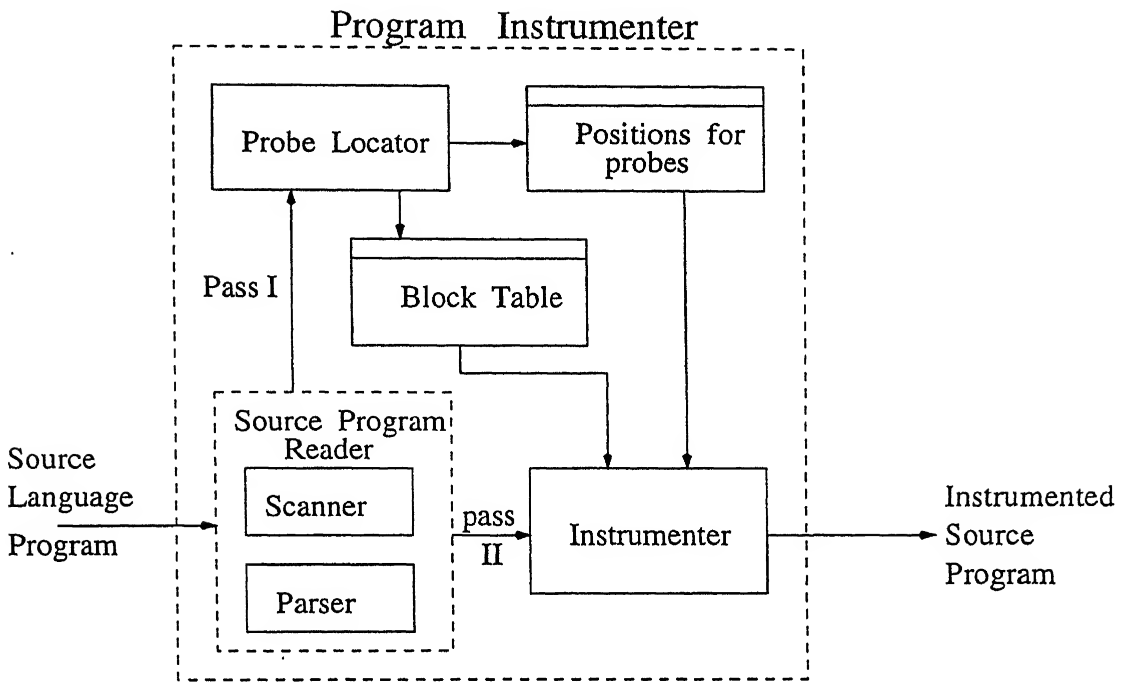

Figure 3.3: Functional Architecture of Program Instrumenter

Source Program reader , reads the source program and provides information regarding the *begining* and *ending* of the blocks, and the location of the probes to be inserted, to the *Probe Locator*. It consists of two components, viz. "scanner" and the "parser". If, the input source program is not syntactically correct, it reports syntax error, and does not allow *instrumenter*, to instrument the source program.

Probe locator constructs *block table*, and also stores the position of the probes in some table, depending upon the information provided by the *Source program reader*. This component operates in the first scanning of the input source program. Structure of the *block table* is shown in figure 3.2.

Instrumenter inserts software probes in the source program, with the help of the *block table* and the table, storing the probe locations. It operates in the second scanning of the source program. The algorithm used by the *instrumenter*, to instrument a program is given in the figure 3.4.

```
Algorithm InstrumentProgram( progfile )
    while (not(end_of_file( progfile )))
        ch = next_char( progfile)
        putcharacter(ch, instrumented_file);
        if(ch == newline )
            if(current_lineno == block_table[currblock].begin_lineno)
                /* For constructs like return statement, goto
                 * statement, procedure call etc., which are not
                 * blocks, begin_lineno and end_lineno
                 * will have the same value
                 */
                cons_type= block_table[currblock].type;
                bline = block_table[currblock].begin_lineno;
                eline = block_table[currblock].end_lineno;
                insert_probe( cons_type, bline, eline);
            endif;
            current_lineno = current_lineno + 1;
            if( nextblock == nil and current_lineno ==
                    block_table[blok_0].end_lineno)
                /* in the main block, put coverage-reporting
                  probes */
                insert_coverage_report_probe(block_table[block_0].type);
                insert_coverage_history_probe(block_table[block_0].type);
            elseif(current_lineno >= block_table[nextblock].begin_lineno)
                currblock = nextblock;
                nextblock = getnext(nextblock);
            endif;
        endif;
    endwhile;
end InstrumentProgram;
```

Figure 3.4: Algorithm for instrumenting source program

3.1.3 Coverage Reporting

The instrumented source program contains probes for

- monitoring the coverage of different constructs,
- maintaining the coverage information across several test sessions, and

- generating the coverage reports.

The instrumented source program gives the coverage statistics for a given set of test cases, with the help of these probes.

Test Coverage Analyzer, generates the supporting routines for the software probes inserted in the source program. These routines includes probes for coverage monitoring, coverage history maintenance, and coverage reporting. Coverage statistics are generated by these routines, which is used to determine the stopping point for software testing operation.

3.1.4 Multi-Session Testing

Software testing operation usually spans for more than one testing session. Each session has different set of test cases, and coverage outcome for each of them may be different. The cumulative coverage outcome of the different test session is an important factor to decide the stopping point. Thus, the Test Coverage Analyzer should maintain the *Coverage History* of these test sessions and report about the cumulative coverage. One simple way to incorporate this feature is to maintain a coverage history file, which stores the cumulative coverage information of all the test sessions. Whenever a new test session is started, the cumulative coverage information from the history file will be loaded and the new values are evaluated using the current coverage. After reporting the coverage statistics, these values will be be written back to the history file.

3.2 Implementation Issues And Decisions

- To make a quality Test Coverage Analyzer for a language, the first requirement is the correct and well-tested grammar for the language. We have used standard and well-tested grammar for Ada'95 available in the public domain. It was a YACC specification file associated with the Lexical analyzer for Ada'95(a Lexfile).

- Another implementation issue was the type of coverage measures to be incorporated in the tool. Although there are several coverage measures(refer to chapter 1), ADACOV implements the following coverage measures.

  - Branch coverage Information(Includes If statement, case statement, while loop, simple loop, for loop, case when components of case statement and exceptions, Ada task, Select Statement, accept statement, goto and return statements)
  - Function coverage,
  - loop coverage for all the loops and tasks,
  - Call coverage,
  - Block-wise coverage statistics,
  - cumulative coverage for the afore-said coverage measures.

- ADACOV supports multi-session testing. Our implementation maintains a History file across various testing session, and at every testing session, the cumulative coverage for all the aforesaid measures are reported in a comprehensive manner(see Appendix B, for sample output); this file is updated at the end of every test session. The history file contains the following information.

  - Type of the statement(e.g. if_statement, while_statement etc.)
  - Line number at which the construct begins.
  - Line number at which the construct ends, and
  - Number of times, it was covered.

3.3 Scope for automatic generation

The objective of this work was to explore the possibilities of automatic development of TCA. Our design shows that the proposed approach can lead to automatic generation of TCA.

In this approach, the only manual phase, is "Instrument Grammar File" shown in the figure 3.1. This is the most crucial phase, and is dictated by the grammar of the language, and the coverage measures to be incorporated. Other phases are same for all languages and can be easily carried out automatically. If we automate this phase, then the whole development process will be an automated one. In other words, automatic generation of TCA, primarily means automating this phase.

Chapter 4

Automatic Generation Of Test Coverage Analyzers

In chapter 3, we have shown that Automatic Generation of TCAs can be achieved by automating the phase "Instrument Grammar" of the design approach. This chapter describes our approach for automating the phase. We also describe various design and implementation issues to accomplish this. The tool, which generates Test Coverage Analyzers, has been named as Generic Test Coverage Analyzer(GCTA). for our discussion in this report.

4.1 A Design Scheme

4.1.1 Design Considerations

- The primary task of the GCTA is to instrument the grammar specification of the source language with probe actions. The nature of these actions are dictated by the design criteria. In our case, these actions store the probe locations in *block table*. In order to instrument the grammar specification file, GCTA should be informed about the productions on which the actions are to be added, and what would be the action. One solution for this, is to use a specification file [GE90]. GCTA uses a specification file, which contains the specifications for the places in which probe actions to be put and what probes are to be put. The syntax and semantics of the specification language is an implementation issue, and is discussed latter in this chapter.

- Our experience with design of Test Coverage analyzer for C and Ada'95 is that, the action part of all the grammar rules is more or less similar with little differences. These differences are dictated by the coverage measures to be incorporated, and semantics of the construct. For example, consider the following production for WHILE statement in C, and the action associated with it.

```
iteration_statement : WHILE '(' expression ')'
   {
     curr_block_depth++;
     block_marker *bmark;
     bmark=new block_marker("{", TCA_LineNum, 1, curr_tab_set,curr_block_depth, WHILE_STMT, 3);
     marker_list.insert(bmark);
     curr_tab_set++;
   }
   statement
     .
     .
```

  The action part of the productions for other statements like *for statement*, *if statement* etc. will be similar to this, except the name of the construct(e.g WHILE_STMT) will be different. Further, these actions involve complex table handling operations, as shown in the example. Our approach hides the complexity of these actions, by providing some specific directives. The probe actions are generated automatically, depending upon the values assigned to these variables/directives. Appendix A describes the detailed usage of these variables and directives. User can directly specify the probes also.

- The current design supports automatic action generation for branch coverage, loop coverage, function coverage, and the Call coverage. If a user wants to incorporate other coverage measures, (s)he has to write actions for it. Our design can be extended to incorporate automatic action generation for new coverage measures. It can be done by adding supporting modules and directives.

4.1.2 Design

■ Functional architecture

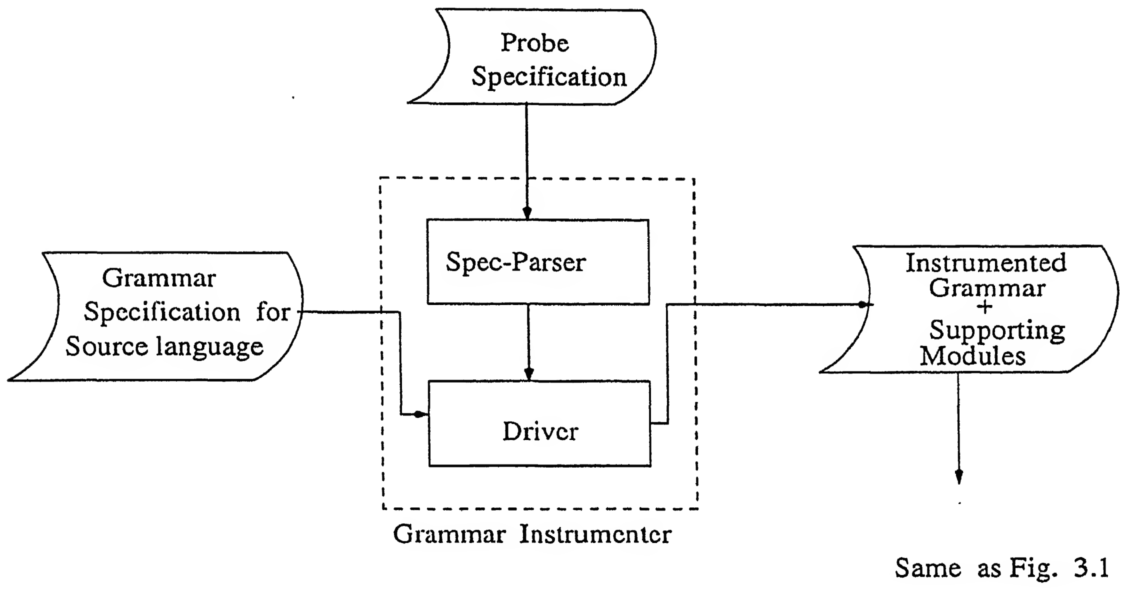
Probe Specification
Spec-Parser
Grammar Specification for Source language
Driver
Instrumented Grammar + Supporting Modules
Grammar Instrumenter
Same as Fig. 3.1

Figure 4.1: Functional diagram of Generic Test Coverage Analyzer

Figure 4.1 shows, the functional diagram of GCTA; it is similar to that of TCA, shown in Figure 3.1, except the new module "Grammar Instrumenter". This module, consists of two components, namely

- spec-parser, and
- driver

Spec-parser reads the probe specification file, checks its syntactic correctness and generates various tables to be used for instrumentation. The probe specification file(refer to Appendix A) contains the production rules of the constructs for which coverage statistics are to be generated, and the specification of the probes. Apart from the symbols in the grammar specification file, a production in the probe specification may contain some extra symbols for specifying the places for probes in a production. *Spec-parser* identifies these symbols, and other GCTA directives, as discussed in the next section. The probe specification file is read prior to reading the grammar specification, which is to be instrumented with appropriate actions.

Driver instruments the grammar specification, depending upon the values in the tables constructed by *Spec-parser*. While instrumenting, it also generates the supporting modules for those actions.

Architecture of *spec-parser*

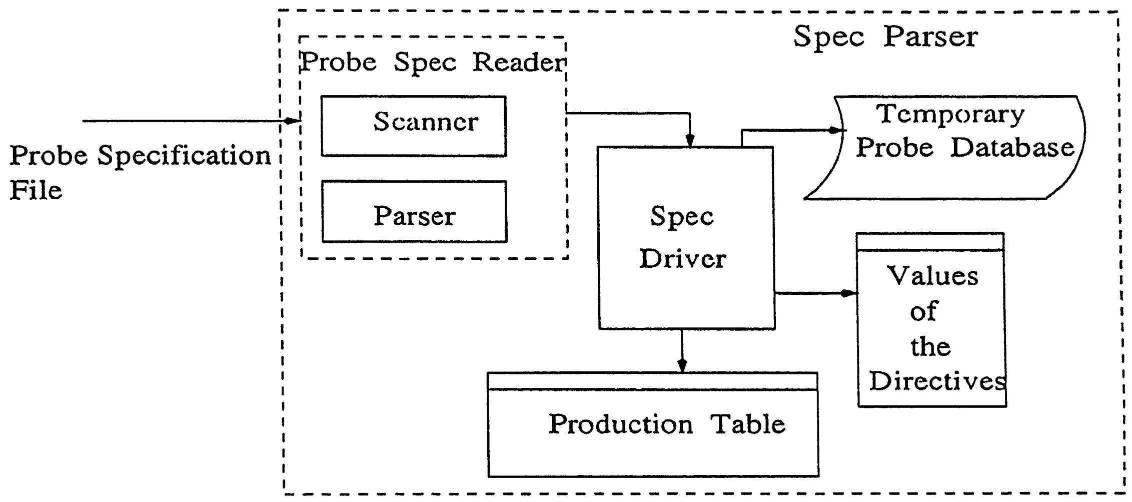

Figure 4.2: Functional architecture of *Spec-parser*

The functional architecture of "Spec Parser" is shown in Figure 4.2. It consists of two modules. They are

- Probe Specification Reader, and
- Spec driver

Probe specification reader , reads the probe specification file, and extracts the values of the different directives and production symbols. If the specification is syntactically correct, it provides these information to *spec driver*. Otherwise, it reports error. The syntax checking is done by the parser component.

Spec Driver , constructs the various tables(refer to section 4.1.3) and temporary databases, depending upon the information provided by the reader. One of the tables is the "production table", which stores the productions to be instrumented. This table stores the rule number, left hand side, and right hand side symbols of each of the production rule. Details of the table management is discussed in section 4.1.3.

The probe specification may contain three types of probes, namely,

- probes to be put into the instrumented grammar specification,
- probes to be put in the instrumented source program, and
- some directives to the driver, to generate some special probes

Spec driver differentiates these probes through some specific GCTA directives(see Appendix A), and stores them in separate temporary files. These temporary files are latter used by the *driver* module of the "Grammar Instrumenter" to instrument the grammar specification, and for generating the supporting modules.

Architecture of *driver*

The *driver* module of the "grammar Instrumenter" consists of four components, as shown in Fig. 4.3. They are,

- Grammar specification reader
- Rule Comparator
- Probe Generator, and
- Instrumenter.

Grammar specification reader reads the grammar specification file, and provides each production rules to the *rule comparator*. The grammar specification file also contains some other informations, like token declaration, variable

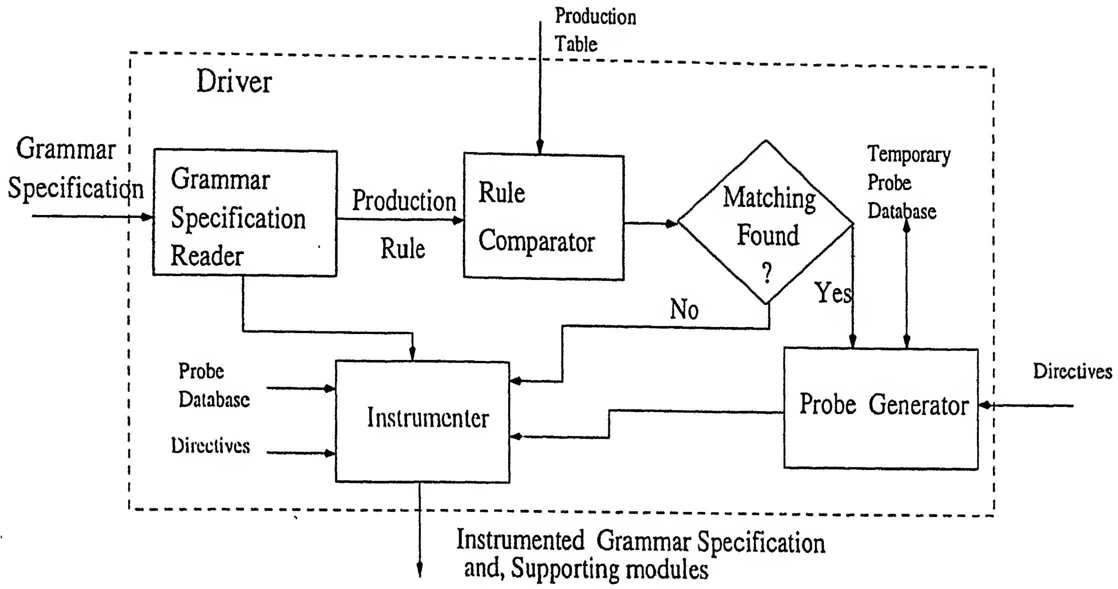

Figure 4.3: Functional architecture of *Driver*

declarations etc., apart from the productions. This information is directly provided to the *instrumenter*, in order to dump them as-it-is(or, with a little bit of customization, as discussed latter) in the instrumented grammar specification file.

The production rules are supplied in the form of two components, viz., lhs(left hand side), and rhs(right hand side) of the rule, to the *rule comparator*. The left hand side is just a character string(because, in Context Free Grammar, lhs of a production cannot contain more than one symbol), and rhs is a linked list of symbols. If the *rule comparator* cannot find any match, the production rule must be dumped as-it-is in the instrumented grammar file. This is done by the *instrumenter*. So, the *grammar specification reader*, provides the rule to the instrumenter as well.

Rule comparator compares each production rule in the grammar specification, with the rules in the probe specification files. The productions in the probe specification file has already been stored in the *production table*(refer to figure 4.6), and the rules in the grammar specification are provided one rule at a time, in the form described above. If a match is found, *rule comparator* provides the rule number of the production to the *probe generator*, otherwise, it returns -1. While comparing the rules, it discards the additional symbols, used in the probe specification for indicating the places of the probes in a production rule.

probe generator generates the action part of the productions, for which match has been found in the probe specification file. It uses the values of the directives, assigned by the user, for generating the action. If no GCTA directive is used to specify the probes in the probe specification, they are treated as the probes to be inserted into the instrumented source program. If the production in the grammar specification file already contains some actions, that is merged with the generated actions. After generating the action, it writes the action into a temporary file. The successive records in the temporary file are separated by some special symbol.

Instrumenter inserts the actions for the productions in the grammar file, for which a match is found in the probe specification file, and generates the instrumented grammar specification file. For accomplishing this task, it reads probe database, and uses the information provided by *specification reader*, *rule comparator*, and the *probe generator*. It also generates, the supporting routines, including the makefile and complete source distribution for the generated Test Coverage Analyzer barring the lexical analyzer.

▮ Algorithms

The algorithms used by the *spec-parser* and the *driver* modules of *program instrumenter* are given in figures 4.4, and 4.5 respectively.

```
Algorithm spec-parser(probe_spec_file)
        error_count = 0;
        Read the probe_spec_file, and while reading,
                Construct production table
                Extract the values of the directives
                differentiate the probes to be written into the
                    instrumented grammar and program files, and store
                    them in temporary files.
                Generate the language dependent probe codes.
        Find useless nonterminals and productions in probe_spec_file.
        If found, report them and increment error_count.
end spec_parser
```

Figure 4.4: Algorithm for spec-parser

4.1.3 Table Management

GCTA internally manages few tables, in order to instrument the grammar file, and for generating the supporting modules. One of these tables is the production table. Each entry in this table contains three fields, namely, the production rule number, pointer to the lhs(left hand side) symbol in the symbol table, and a pointer to a list storing the right hand side(rhs) of the production. This list stores the pointers to the symbols in the rhs of the rule. Figure 4.6 shows the structure of this table. Symbol table stores the information about all the symbols in the probe specification file. GCTA needs the name of the symbol only. The implementation detail of the production table is given Appendix B.

Other tables include

- a table for storing the probe nonterminals(these are the extra symbols inserted in a production in the probe specification file, to indicate the places for the probes),
- a table for storing the construct names to be used for coverage reporting,
- a table for storing the index of the probes stored in the temporary databases, for each production rule to be instrumented.

```
Algorithm driver(gram_spec_file)
    if (error_count > 0)
        /* If there is any error in the probe specification,
         * do not instrument
         */
       return
    endif;
    put_instrument_header(ins_gramfile);
        /* Declare the global variables, data types etc. in the instru-
         * mented grammar file, that will be required by the actions
         */
    while(not(end_of_file(gram_spec_file)))
        current_section = check_section(gram_spec_file);
        if (current_section <> grammar_rule_section)
            copy_section(current_section, ins_gramfile);
        else /* read and instrument productions */
            while (not(end_of(grammar_rule_section)))
                curr_prod = get_production(gram_spec_file);
                ruleno = compare_rule(curr_prod, prod_table);
                if(ruleno < 0)
                    /* If no match is found */
                    copy_production(curr_prod, ins_gramfile);
                else
                    generate_action();
                    instrument_production(ruleno, ins_gramfile);
                endif;
                merge_action_part(gram_spec_file, ins_gramfile);
                  /* If the grammar specification, contains some action
                   * associated with the current production, merge(copy, in
                   * case, no matching is found)it
                   */
            endwhile;
        endif;
    endwhile;
     /* generate supporting source distribution */
    generate_supporting_routines();
    generate_makefile();
End driver;
```

Figure 4.5: Algorithm used by the driver

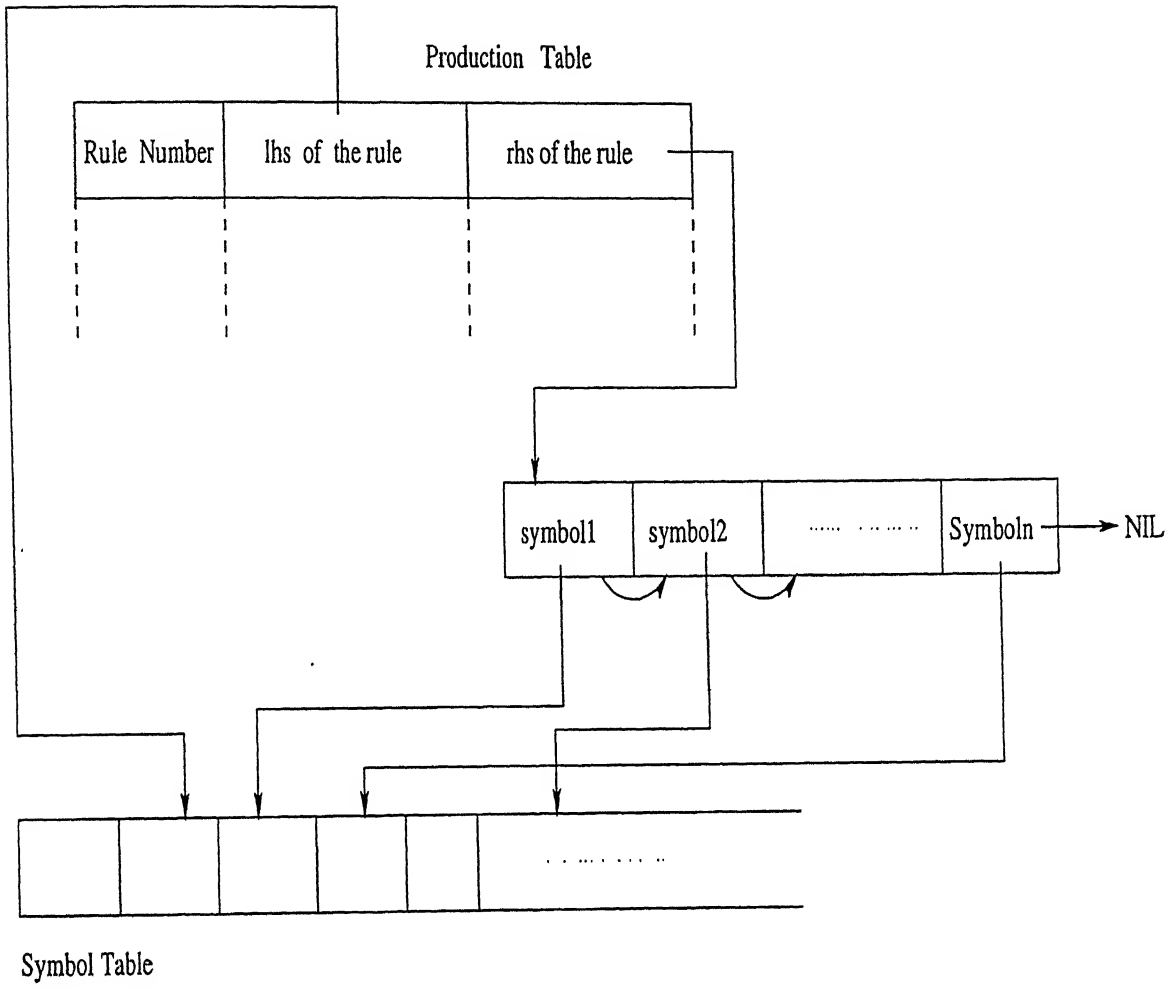

Figure 4.6: Structure of the production table

- a table for storing the probes to be inserted into the final instrumented program, when a particular construct is scanned by the TCA. This table is required, because the source program is instrumented by the generated Test Coverage Analyzer, not by GCTA. This table is dumped(in the form of a table of structure) in a file of the output source distribution, along with a supporting module. This module reads this table and inserts the probes into the instrumented source program.

All these tables are basically linked lists with one or two fields.

4.1.4 Output Of GCTA

The output of GCTA comprises of the instrumented grammar file, and the complete source distribution barring the lexical analyzer for the language for which the Test Coverage Analyzer is generated. The generated source distribution follows the design described in chapter 3. GCTA also generates the makefile for this distribution, leaving the entry for Lex-file blank. User has to make a little modification in the makefile and in the lex-file. The details can be obtained from Appendix A.

4.2 Implementation Issues And Decisions

- The first implementation issue was about the syntax and semantics of probe specification language. If the specification could be written in some standard specification language, user will require minimum effort to learn it. After reviewing several specification language used by standard tools like LEX [Les75], YACC [Joh75], etc. we decided to accept the language used by YACC as the specification language for the GCTA. The primary reason for this selection is that, the probe specification contains some production rules, actions, and some language specific codes. A YACC specification file may contain all these, and it is a popular specification language for specifying grammar rules.

- The next implementation issue was about the parser(i.e *Spec-parser*) for the specification language. We have used BISON (GNU YACC)[1] to parse the probe specification file, because it is a public domain software and the source codes are easily available. We used BISON as a module, which checks the specification file for correctness and provides the necessary tables to the *driver*. The production table, discussed in the section 4.1.3, is constructed by BISON. The *driver* uses this table while instrumenting the grammar specification (which is a BISON specification file).

- Our design approach makes an attempt to minimize the user efforts in specifying the actions for the productions in the probe specification. The mechanism

[1]Source distribution can be obtained from http://sunsite.unc.edu/pub/gnu/bison-1.25.tar.gz

is to introduce some GCTA variables and directives, as discussed in the section 4.1. The syntax for specifying these directives/variables should not violate the syntax of the specification language. We decided to use these directives in some specific section of the probe specification. The details about these directives/variables are listed in Appendix A.

- An action in the probe specification file may contain three types of probes discussed earlier. How to distinguish them, is a major implementation issue. Our implementation uses the convention that any action statement without some special directive means that, it is to be inserted in the final instrumented program.

- Another question that came to our mind was about the additional nonterminals used for specifying location of probes in production rules. Here also, we decided to use a GCTA directive for differentiating these nonterminals, with those in the grammar specification.

- Once a program is instrumented with user defined probes, user would like to provide codes for them. For example, suppose a user specifies a function/procedure call in a probe; then user should be able to define that function/procedure in the source language of the program. We decided to allow users to write such codes after the probe specification in the spec-file (i.e after the 2nd "%%"). Some specific GCTA directives(see Appendix A) are also provided to customize these codes.

- Another implementation issue was about the target language in which the grammar actions, and supporting functions are to be generated. Our prior implementation of Test Coverage Analyzer for Ada95 was done in C++ Programming Language [Str94], and we found it convenient to handle complex data structures. Thus we decided to use C++ as the target language for GCTA.

Chapter 5

Results

5.1 Development Environment

Both the tools(i.e. ADACOV and GCTA) have been developed and implemented under the following environment.

Machine	:	DEC Alpha, 2000 4/233 server
Operating System	:	DEC OSF/3.0
Language	:	C++ and C
Utilities	:	BISON, FLEX, Make

ADACOV has been developed in C++ language, and GCTA in C. The total source distribution contains around 6500 Lines of Code for ADACOV, and around 12000 Lines of Code for GCTA.

5.2 ADACOV

5.2.1 Salient Features

- ADACOV can be used as Test Coverage Analyzer, with the coverage measures stated in chapter 3.
- ADACOV can be used as a beautifier for Ada'95 program.

- The instrumented program can be stripped off the probes inserted, to get the source program.

Appendix B gives complete user's manual for ADACOV.

5.2.2 Application Experiences

ADACOV has been tested with a variety of test programs ranging from 50 lines to 1000 lines of code in size, and it has been found working properly. Apart from TCA, ADACOV incorporates a beautifier for Ada95. The instrumented program is generated with proper indentation. The Coverage Reports are generated in easy to read format. A typical format of a coverage report is given in Appendix B.

Presently, ADACOV can be used to test Ada programs in a single file. The capability for instrumenting all the files included in the form of a package is not implemented.

5.3 GCTA

5.3.1 An Example

GCTA generates Test Coverage Analyzer for a language, upon receiving the grammar and probe specification. A sample specification file is shown in the Appendix A. The generated Test Coverage Analyzer, instruments any source program written in that language. For example, code segment given below, shows codes inserted by the generated test coverage analyzer, in an Ada program.

```
with coverage;
  .
  .
with Calendar;
package body Room is
      .
      .
   task body Maitre_D is
      .
      .
   begin
```

```
    IF coverage.C_O_VERAGE then
       coverage.update_coverage(coverage.getarrayname(TASK_BODY), 58, 191);
    End If;
       .
       .
    for Which_Window in Phil_Windows'range loop
       IF coverage.C_O_VERAGE then
          coverage.update_coverage(coverage.getarrayname(FOR_LOOP), 100, 110);
       End If;
       .
       .
    end loop;
       .
       .
    loop
       IF coverage.C_O_VERAGE then
          coverage.update_coverage(coverage.getarrayname(SIMPLE_LOOP), 120, 189);
       End If;
         .
         .
       If coverage.C_O_VERAGE Then
           coverage.UpdateCoverage(caseArray, 128, 181);
       End If;
       case State is
         when Phil.Breathing =>
             If coverage.C_O_VERAGE Then
                 coverage.UpdateCoverage(CaseWhen, 129, 133);
             end If;
           .
           .
       end case;
         .
         .
     end loop;
       .
       .
  end Maitre_D;
     .
     .
end Room;
```

In this code segment, the statement "with coverage;", is added by the TCA. The file "coverage.ads" contains the data structure and function specification of the codes to be used for instrumentation. As, this is a language dependent matter, user has to provide code for this. GCTA provides some facilities for customizing such code(Refer to Appendix A). All statements, included within If `coverage.C_O_VERAGE` `then ... End If;` also inserted by TCA. In fact, these statements monitors coverage.

5.3.2 Performance Evaluation

We have examined the performance of GCTA by testing GCTA-generated TCAs, on various programs, written in Ada'95 and C. For comparing the performances, we have used two hand coded Test Coverage Analyzers for C and Ada'95 respectively. The TCA for C was developed earlier [Pal93], and that for Ada'95(ADACOV) has been developed by us.

For generating the TCAs for C and Ada'95, by using GCTA, we have used standard, and well-tested YACC and LEX specification files available in the public domain. The same specification files were used for hand-coded TCAs also. This assures the correctness of the parser, and scanner components in both types of TCAs(i.e. hand-coded and generated).

TCA for (hand coded)	No. of lines[1] (code)	Development Period (code)	TCA for (generated)	No. of lines [2] (specs)	Development Period (specs)
C	5400	-	C	175	2 days
Ada95	4251	60 days	Ada95	165	2 days

Table 5.1: Experimental Results

The sizes of the programs, used for testing purpose varies from 50 lines to 1000 line of codes, and GCTA-generated TCAs were found to be working properly. The source program instrumented by GCTA-generated TCA, is at par with that instrumented by hand coded TCAs. On the other hand, it has been observed that GCTA

[1]Excluding the no. of lines in LEX and YACC specification files (without any action). Also, excluding the no of lines for the language specific probe codes.

[2]No. of lines in the probe specification file. The user defined language specific probe codes are not counted here. The sizes of LEX and YACC files are also not counted here, because they are standard for a particular language, and can be directly obtained from the public domain.

requires minimal information to generate the TCAs. A specification file of size 150-200 lines is enough for generating Test Coverage Analyzers for a language. Table 5.1 summarizes the experimental results.

These experimental results have shown that, using GCTA, one can develop reliable Test Coverage Analyzers at a rate much faster than(about 25 to 35 time) hand coded development. The probe specification contains minimal information, and are easy to specify. The specification language is found to be flexible enough to incorporate generated and user defined probes effectively.

GCTA-generated TCA for	Size of Source distribution in lines of code
Ada'95	5048
C	5368

Table 5.2: Sizes of the source distribution for GCTA-generated TCAs

We have also analyzed the sizes of the source distribution for the TCAs, both hand-coded and generated. Table 5.1 shows the the sizes of the source distribution for hand coded TCAs. Table 5.2 shows the same statistics for GCTA-generated TCAs. Note that, ADACOV(hand-coded TCA for Ada'95) incorporates a beautifier for Ada'95, which is not there in GCTA-generated TCAs.

Chapter 6

Conclusion And Future Work

The main objective of this thesis work was to automate the development of Test Coverage Analyzers. In this thesis, we looked into the problems associated with automation. To achieve the goal, we started with a survey of some of the existing Test Coverage Analyzers and the current works on this field. We also analyzed the status of our work, i.e. where does it stand, and its relevance in the software products domain.

To make a process automatic, it is important to understand the existing manual process first. That is why, we looked into the details of the design approach for Test Coverage Analyzers. We have proposed a design procedure, and we have found our approach, a language independent one.

The proposed design scheme has been thoroughly tested for its correctness, by applying it to design and implement a Test Coverage Analyzer for Ada'95(ADACOV). ADACOV had been tested for its performance, over a variety of Ada programs, and it was found to be quite satisfactory.

Our goal was to automate the development of Test Coverage Analyzers. From our design experience, we found that, it is basically a task of automating three steps(refer to chapter 3). A design scheme along with various implementation issues for accomplishing this task has been discussed in this thesis. We implemented a tool, which is capable of generating Test Coverage Analyzers for a language, after reading certain specification. The application experiences have shown that, it can increase the productivity tremendously, compared to hand-coded development.

6.1 Future Work

GCTA can automatically generate probe actions, depending upon the values assigned to some specific directives(see Appendix A). This reduces the user's effort in writing probe actions. However, the directives in the current version can support automatic generation of actions related to the following coverage measures only.

- Branch Coverage
- Call Coverage
- Function Coverage
- Block-wise coverage.

If someone wants to incorporate some other measure, user has to explicitly specify the action. Thus, one direction for further work could be incorporating automated action generation feature for other coverage measures like relational operator coverage, race coverage etc.

This work can be extended to incorporate automatic generation of other Software Engineering tools like Test data generators, static analyzers etc. Test Data Generation problem more or less resembles coverage analysis. An approach for automatic test data generation can be found in [Kor90]. Some work on this field can be found in [Pal93].

Bibliography

[Aga94] Hiralal Agarwal. Dominators, superblocks and program coverage. In *21st ACM SIGPLAN-SIGACT, Symposium on Principles of Programming languages*, pages 26–37, January 1994.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles Techniques and Tools.* Addition weseley, Reading, MA, 1986.

[Bei90] Boris Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New York, second edition, 1990.

[Cor96] Steve Cornett. Software test coverage analysis. Bullseye testing Technology, 1996. URL http://www.bullseye.com/coverage.html.

[GE90] J. Grosch and H. Emmelmann. A toolbox for compiler construction. In *Lecture Notes in Computer Science No. 477*, pages 22–24. Springer-verlag, October 1990.

[How82] Wiliam E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, SE-8(4):371–379, July 1982.

[Jal91] Pankaj Jalote. *An Integrated Approach to Software Engineering.* Springer-Verlag, New York, 1991.

[Joh75] S. C. Johnson. Yacc - yet another compiler compiler. Technical Report 32, Bell telephone Laboratories, Murray Hill, NJ, 1975. UNIX programmers manual, Vol II.

[KHZ82] Uwe Kastens, Brigitte Hutt, and Erich Zimmermann. The compiler generator gag. In *Lecture Notes in Computer Science No. 141*. Springer-verlag, October 1982.

[Kor90] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.

[Les75] M.E. Lesk. Lex - a lexical analyzer generator, computer science technical report. Technical Report 39, Bell telephone Laboratories, Murray Hill, NJ, 1975. UNIX programmers manual, Vol II.

[Mar96] Brinn Marick. A tutorial introduction to gct. Testing Foundations, 1996. Anonymous FTP site ftp://www.cs.uiuc.edu/pub/testing/gct.files.

[Mor90] Larry J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, August 1990.

[Mos90] H. Mossenback. Coco/r - a generator for fast compiler front ends. Technical report, Department Infomatik, Institut fur Computersysteme, ETH Zurich, 1990. Anonymous FTP site ftp://ftp.fit.qut.edu.au/pub/coco.

[Mye79] Glen Myers. *The Art of Software Testing*. Wiley Interscience, New York 1979.

[Pal93] Atul S. Paldhikar. Coverage based testing and test data generation. Master's thesis, Department of Computer Science, IIT Kanpur, January 1993 No. MT-CS-93-03.

[PD93] Ruben Prieto-Diaz. Status report- software reusability. *IEEE Software* pages 61–66, May 1993.

[Pro82] Robert L. Probert. Optimal insertion of software probes in well-delimite programs. *IEEE Transactions on Software Engineering*, SE-8(1):34–4 January 1982.

[RKC75] C. V. Ramamoorthy, K.H. Kim, and W. T. Chen. Optimal placement of software monitors aiding systematic testing. *IEEE Transactions on Software Engineering*, SE-1(4):403–411, December 1975.

[Sha91] S. Sharma. Ccov a coverage based testing tool for c programs. Master's thesis, Department of Computer Science, IIT Kanpur, January 1991.

[Str94] Bjarne Stroustrup. *The C++ Programming language.* Addition weseley Publishing Company, AT & T Bell Laboratories, Murray Hill NJ, second edition, 1994.

[Whe96] David A. Wheeler. Lovelace tutorial version 5.6. An Ada95 Tutorial, 1996. URL http://www.adahome.com/Tutorials/lovelace.html.

Appendix A

User's Manual for GCTA

This document is meant for users of gcta. The document begins with a detail discussion of the format for the probe specification file, followed by a comprehensive listing of gcta variables/directives and their usage. This document also describes the command synopsis, and explains the various options and the environment variables. Finally, a sample example is given in section A.4.

A.1 Syntax for Probe Specification

Figure A.1 shows the structure of a probe specification file, used by the Generic Test Coverage Analyzer(GCTA). The format is exactly same as a YACC specification file. It has four sections, viz.

Pre-declaration section : In this section, any declaration can be made. But GCTA looks for the directive "TCA_probes", which specifies the probe non-terminals used in the specification file. Other informations/declarations in this section carries no meaning to GCTA.

Token Specification : This section is meant for the token specifications (like YACC specification file). Any symbol, for which, no production rule is defined, must be declared as token in this section. Otherwise, GCTA does not instrument the input grammar specification file. The start symbol specification should also be made in this section. Other declarations (e.g. type, associativity etc.) are exactly same as a typical YACC specification, but have nothing to do with GCTA.

Grammar Rule Section : This section contains the actual grammar rules and the probe actions. The grammar rules may contain additional probe nonterminals and their productions. These nonterminals must be declared in the "Pre-declaration Section". Otherwise, GCTA will not be able to match the corresponding rule in the grammar specification file, if these nonterminals appear in a rule of the probe specification file. The probe specification, is same as usual action specification in a YACC file. The probe, specification may contain several GCTA directives. A detail discussion of these directives are given in section A.2.

Post declaration section : In this section, user may define their language specific probe codes. GCTA provides some specific language directives(refer to section A.2 which enables user to customize their probe codes.

The syntax for all these sections, are exactly same as that of a typical YACC specification file.

A.2 GCTA directives

A probe action may be meant for the instrumented grammar specification, and/or for the final instrumented grammar. It has been observed that action part of the probe specification involves lot of table handling operations, and they are more or less similar with little or no differences. So these probes action can be generated with little informations; this would also hide the complexity of the probe actions. Further, as stated in the previous section, the probe specification file must clearly distinguish the additional probe nonterminals with those in the grammar specification. Moreover, in the post declaration section, user might like to customize their codes, and might like to use some input program dependent values. All these can be done easily with the help of some specific GCTA variables and directives. This section enumerates these variables/directives and their utilities.

associativity etc.) are exactly same as a typical YACC specification, but have nothing to do with GCTA.

Grammar Rule Section : This section contains the actual grammar rules and the probe actions. The grammar rules may contain additional probe nonterminals and their productions. These nonterminals must be declared in the "Pre-declaration Section". Otherwise, GCTA will not be able to match the corresponding rule in the grammar specification file, if these nonterminals appear in a rule of the probe specification file. The probe specification, is same as usual action specification in a YACC file. The probe, specification may contain several GCTA directives. A detail discussion of these directives are given in section A.2.

Post declaration section : In this section, user may define their language specific probe codes. GCTA provides some specific language directives(refer to section A.2 which enables user to customize their probe codes.

The syntax for all these sections, are exactly same as that of a typical YACC specification file.

A.2 GCTA directives

A probe action may be meant for the instrumented grammar specification, and/or for the final instrumented grammar. It has been observed that action part of the probe specification involves lot of table handling operations, and they are more or less similar with little or no differences. So these probes action can be generated with little informations; this would also hide the complexity of the probe actions. Further, as stated in the previous section, the probe specification file must clearly distinguish the additional probe nonterminals with those in the grammar specification. Moreover, in the post declaration section, user might like to customize their codes, and might like to use some input program dependent values. All these can be done easily with the help of some specific GCTA variables and directives. This section enumerates these variables/directives and their utilities.

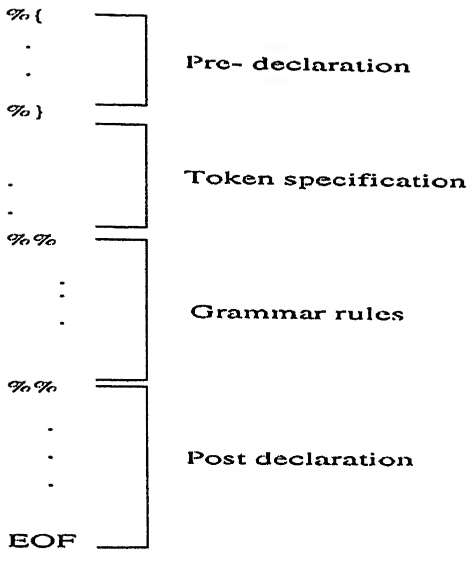

Figure A.1: Format of Probe Specification File

GCTA has two type of directives, viz. directives/variables for a) pre-declaration and grammar rule section, and for b) post declaration section(ref. section A.1). The first type of directives start with the prefix "TCA_", and the second type is prefixed by the character "$".

A.2.1 Directives for predeclaration and grammar rule section

The directives/variables, that can be used in these sections are described in the table A.1. The predeclaration section can use only one GCTA directive, which is "TCA_probes". Other directives does not carry any meaning to this section. Again, this directive(i.e TCA_probes) cannot be used in the grammar rule section

Table A.1: GCTA variable/directives for Predeclaration and Grammar Rule section

Var./Directive name	**Meaning and syntax**
TCA_prname	This variable defines the name of a construct. The value assigned to this variable, is used by GCTA to define the name of the current block. This is an essential variable for generated probes. Otherwise, GCTA cannot generate probes. This value is used to count the occurrence of a particular type of construct in the source program, and also to decide which probes to be put in the instrumented program, by the "instrumenter". GCTA defines a standard probe name, viz. *NONEXEC*, which means that the particular block is a nonexecutable one(like declaration block). Syntax : TCA_prname = Name_of_the_Construct; Example : TCA_prname = IF_STMT; This means that the name of the current construct is "IF_STMT". This variable can be used in the action part of a rule in *Grammar rule* section only.
TCA_prtype	This variable is used to indicate the "beginning" and "ending" of a construct. It can have either of the two GCTA-defined values, viz BEGINING, and ENDING. Syntax : TCA_prtype = GCTA-defined constants; Example : TCA_prtype = BEGINING; This indicates the begining of the construct, named by TCA_prname. The value assigned to this variable, anlong with the value of TCA_prname, determines, what probes to be put and where. This variable can be used in the action part of a rule in *Grammar rule* section only.

Var./Directive name	Meaning and syntax
TCA_tmpname	It is like the variable TCA_prname, but is used to store the probe-name temporarily, and the value assigned to this variable is not communicated to GCTA. Finally, this variable should be assigned to TCA_prname. Basically, this variable is used to transfer the name of a construct, across production rules(see example in Section A.4). Syntax : TCA_tmpname = name_of_the_construct; Example : TCA_tmpame = SIMPLE_LOOP; This variable can be used in the action part of a rule in *Grammar rule* only.
TCA_probes	This variable informs GCTA about the additional probe nonterminals. Basically, this is a list, storing the name of the probe nonterminals. This specification is very much important for GCTA. Otherwise, it cannot differentiate the probe nonterminals with those in the grammar specification. So, if a probe nonterminal appears in a production in the grammar rule section, GCTA will not be able to find a match in the grammar specification, resulting improper instrumentation of the grammar specification. Syntax : TCA_probes = { nonterm1, nonterm2, ... }; Example : TCA_probes = { if_beg, if_end, else_beg, else_end }; This variable can be used in *Predeclaration section* only.

Var./Directive name	Meaning and syntax
TCA_beginline	This is a directive to the GCTA, which directs GCTA to replace this value with the begin line number of the block(specified by TCA_prname, and TCA_prtype). Thus, whenever GCTA finds the occurrence of this directive, it replaces it with the beginning line number. This directive is meant for the probes to be written into the final instrumented program. Syntax : TCA_beginline Example : UpdateCoverage(Casearray, TCA_beginline, ...) GCTA will repace TCA_beginline with the begin line number of the block/statement for which this probes to be put. This directive is meant for the action part of the *Grammar rule* section only.
TCA_endline	This directive is same as the previous one, except, it directs GCTA to put the ending line number of the statement/block. Syntax : TCA_endline Example : UpdateCoverage(Casearray, TCA_beginline, TCA_endline) This directive is meant for the action part of the *Grammar rule* section only.

Var./Directive name	Meaning and syntax
TCA_LineNum	This variable is internally maintained and updated by GCTA, to keep count of the current source line number. To provide some additional flexibility, user is also allowed to assign values to this variable. The valid values for this variable can be any positive number, or special GCTA values "INC" or "DEC". INC, means increment and DEC means decrement by 1. <u>Syntax :</u> TCA_LineNum = value; <u>Example :</u> TCA_LineNum = 7; or TCA_LineNum = INC; or TCA_LineNum = DEC; This variable is meant for the action part of the *Grammar Rule* section only.
TCA_dump	Sometimes, user may need to put some actions into the Instrumented grammar specification file directly(i.e. not generated by GCTA). To do that, user must specify such action through this directive. Any action specification, without this directive, and the assignments for the variables TCA_prname, TCA_prtype, TCA_LineNum, and TCA_tab is meant for the final instrumented program. <u>Syntax :</u> TCA_dump{ ... action statements ...} <u>Example :</u> TCA_dump{ $for(i = 0; i < k; i++)$ ... } This directive is meant for the action part of the *Grammar Rule* section only.

Var./Directive name	Meaning and syntax
TCA_tab	This variable is kept for setting the tab values for a construct. This is required for beautification. Currently, generic beautifier is not implemented. Thus this variable does not carry any meaning. The syntax and values are exactly same as that of TCA_LineNum This variable is meant for the action part of the *Grammar Rule* section only.

A.2.2 Directives For The Post Declaration Section

The variables/directives used in this section are used for customizing the user defined probe codes. Unlike, the variables/directives in the Predeclaration and Grammar Rule section, they begin with a dollar sign. Table A.2 summarizes these variables/directives.

Table A.2: Variables/directives used in Post declaration section

Var./Directive name	Meaning and syntax
$filename	While writing the language specific probe codes, user may wish to specify the name of the files in which these codes to be written. That is exactly what user can do through this variable. User can assign the name of the file in which the probe codes to be written, by assigning the name of the file to this variable. Everything, following this assignment, (up-to $endfile or EOF) will be written into that file. Otherwise, the probe codes will be appended to the instrumented grammar file. Syntax : $filename = name_of_the_file Example : $filename = coverage.ads

Var./Directive name	Meaning and syntax
$endfile	This directive is complementary to the previous directive(i.e. $filename). This directive indicates that the codes following this are not to be written into the file specified by the previous assignment to "$filename". Syntax : $endfile Example : $endfile
$count	It has been observed that, some values in the probe codes are dependent upon the input source program. For example, the number of a particular construct, no of blocks etc. These values can be obtained, only when GCTA scans the source program. To incorporate such values, GCTA provides this directive, which means that GCTA has to replace it with the proper values. Syntax : $count(name_of_the_construct) The name_of_the_construct, must be one of the values assigned to TCA_prname, in the Grammar Rule section. Example : CoverageTable(1 .. $count(IF_THEN));
$bcount	This directive is similar to the previous one, except it is used to count the total number of blocks in the source program. Syntax : $bcount Example : Blocktable(1..$bcount)

A.3 How to use GCTA?

GCTA reads two specification files, viz. the *probe specification* and the *grammar specification*. As states earlier, both the files are YACC specification file. The structure of the probe specification file has already been discuessed in section A.1. This section describes the usage of GCTA.

A.3.1 Command Synopsis

```
gcta [-v] [-norg] probe_specification_file [ -i grammar_specification_file]
```

<u>**Description**</u>

User must provide the name of the probe specification file, in the command line. Other optional arguments are described below:

-v This flag directs gcta to operate in the verbose mode. While operating in this mode, gcta displays all the intermediate messages in the standard error output.

-norg This flag indicates that the original actions in the grammar specification file should be removed. By default, GCTA keeps those actions.

-i grammar_specification_file This option specifies the name of the grammar specification file. If this is not specified, gcta prompts for this file, after scanning the probe specification file.

Upon the successful completion of the processing, gcta produces the instrumented grammar specification file, several utility files, and the make file for the generated coverate analyzer. The name of the generated makefile is "covmake". In this makefile, the entry for the LEXFILE.l is kept blank. So user has to enter the name of the LEX specification file here, and also he/should make the following updates in the LEX specification file:

1. Declare TCA_LineNum as "extern int TCA_LineNum;", in the declaration section.

2. Increment TCA_LineNum, by one, whenever a newline is scanned. This value is used to identify a particular construct in the source program.

Example

Here is a sample run of the command.

```
<75>csealpha3:ada%gcta -v -i ada_grammar8x.y covspec.y


gcta version 1.1, Developed by Utpal Bhattacharyya
covspec.y :Warning : You did not use TCA_prtype and TCA_prname for a probe action.
Don't know how to generate probe at line 62
        .
covspec.y :Warning : You did not use TCA_prtype and TCA_prname for a probe action.
Don't know how to generate probe at line 129
No errors detected.
Instrumenting yacc file ada_grammar9x.y ....
Following rule(s) in specfile covspec.y has no correpondant in file ada_grammar9x.y
          covbegin : if_stmt
          covbegin : case_stmt
          covbegin : loop_stmt
          covbegin : block
          covbegin : task_body
          covbegin : subprog_spec
generating supporting functions ....
Generating makefile ...
The   grammar file has been instrumented; some of the supporting routines  are also generated.
The name of the makefile generated is "covmake". To make  the   whole distribution workable,
do the following updates :
   1. In the makefile set the variable LEXFILE.l to your lexfilename
   2. Declare the TCA variable TCA_LineNum, as "extern int TCA_LineNum;" in the lexfile
   3. Increment this variable, whenever you scan a newline
```

Do not worry about the Warnings. These warnings are just to remind the users that, in those lines, probe actions are specified, but not enough information is given for generating the probes.

A.3.2 Environment Settings

GCTA generates some temporary files, while scanning the probe specification file. Normally these files are generated in /tmp directory. However, user may specify the directory for the temporary files by setting the environment variable "GENCOVTMP". The temporary files are removed at the end of processing.

A.4 A Sample Example Of The Probe Specification File

Here is a typical example of the probe specification file:

```
%{
        /** Pre-declaration section **/
        TCA_probes = { ifelse_probe, else_probe , loop_beg, loop_end, blk_probe};
%}
/****** Token declaration ***/
%token cond_part statement_s IF END ELSIF IF ELSE FOR designator  identifier
%token WHILE IN BEGiN condition PROCEDURE compound_name decl_part alternative_s CASE block_decl
%token formal_part_opt FUNCTION formal_part_opt RETURN name  handled_stmt_s case_hdr pragma_s
%token label_opt basic_loop id_opt reverse_opt discrete_range TASK BODY simple_name IS
%start covbegin
%%

/*** Grammar Rule Section ***/
covbegin :
if_stmt
| case_stmt
| loop_stmt
| block
| task_body
| subprog_spec
;

if_stmt :
IF cond_clause_s else_opt END IF ';'
;

cond_clause_s :
cond_clause
| cond_clause_s ELSIF cond_clause
;

cond_clause :
cond_part ifelse_probe statement_s
{
        TCA_dump {
                printf("cod_part \n");
        };
        TCA_prtype = ENDING;
        TCA_prname = IF_THEN;
};
```

```

loop_stmt :
label_opt iteration loop_beg basic_loop id_opt ';' loop_end
;

iter_part :
FOR identifier IN
{
        TCA_tmpname = FOR_LOOP;
};

iteration :
{
        TCA_tmpname = SIMPLE_LOOP;
}
| WHILE condition
{
        TCA_tmpname = WHILE_LOOP;
}
|iter_part reverse_opt discrete_range
;

case_stmt :
{
        TCA_prtype = BEGINING;
        TCA_prname = CASE_STMT;
        TCA_LineNum = DEC;
        If coverage.C_O_VERAGE Then
            UpdateCoverage(caseArray, TCA_beginline, TCA_endline);
        End If;
}
case_hdr pragma_s alternative_s END CASE ';'
{
        TCA_prtype = ENDING;
        TCA_prname = CASE_STMT;
        TCA_LineNum = INC;
};

block :
label_opt block_decl
{
        TCA_tmpname = SEQ_STMTS;
}
block_body END id_opt ';'
;

block_body :
BEGiN blk_probe handled_stmt_s
```

```
{
        TCA_prtype = ENDING;
        TCA_prname = TCA_tmpname;
};

subprog_spec :
PROCEDURE compound_name formal_part_opt
{
        TCA_tmpname=PROC_BODY;
}
| FUNCTION designator formal_part_opt RETURN name
{

        TCA_tmpname = FUNC_BODY;
}
| FUNCTION designator
{
        TCA_tmpname = FUNC_BODY;
};

task_body :
TASK BODY simple_name IS decl_part
{
        TCA_tmpname = TASK_BODY;
}
block_body END id_opt ';'
;

blk_probe :
{
        TCA_prtype = BEGINING;
        TCA_prname = TCA_tmpname;
        IF coverage.C_O_VERAGE then
           update_coverage(getarrayname(TCA_tmpname), TCA_beginline, TCA_endline);
        End If;
};

loop_beg :
{
        TCA_prtype = BEGINING;
        TCA_prname = TCA_tmpname;
        IF coverage.C_O_VERAGE then
            update_coverage(getarrayname(TCA_tmpname), TCA_beginline, TCA_endline);
        End If;

};

loop_end :
```

```
{
        TCA_prtype = ENDING;
        TCA_prname = TCA_tmpname;
};

else_opt :
| ELSE else_probe statement_s
{
        TCA_prtype = ENDING;
        TCA_prname = IF_ELSE;
};

ifelse_probe :
{
        TCA_prtype = BEGINING;
        TCA_prname = IF_THEN;
        IF coverage.C_O_VERAGE THEN
            Ada.Text_IO(item=>"IF Statement");
        ENDIF
};

else_probe :
{
        TCA_prtype = BEGINING;
        TCA_prname = IF_ELSE;
        IF coverage.C_O_VERAGE THEN
            Ada.Text_IO(item=>"IFELSE Statement");
        ENDIF
};

%%
/*** Post Declaration section **/

$filename = coverage.ads
With Ada.Text_IO;
Package coverage is
  Subtype CoverageRange is Integer range 0..Integer'Last;
        Type Coverage_info is
                record
                        CoverageStatus   : Integer;
                        begin_Lineno     : Integer;
                        end_Lineno       : Integer;
                        Times            : Integer;
                end record ;
        type CoverageTable is array ( CoverageRange range <> )
        of Coverage_info ;
        type BlockTable is new  CoverageTable;
        -- Constats
        M_forstmt    : Constant Integer := $count(FOR_LOOP);
        M_simpleloop : Constant Integer := $count(SIMPLE_LOOP);
        M_casestmt   : Constant Integer := $count(CASE_STMT);
        M_taskstmt   : Constant Integer := $count(TASK_BODY);
        C_O_VERAGE     : Constant Integer := 1;
        .
        .
        ForStmt_Coverage     : CoverageTable(1..M_forstmt);
        SimpleLoop_Coverage  : CoverageTable(1..M_simpleloop);
        CaseWhen_Coverage    : CoverageTable(1..M_casewhen);
        Block_Coverage       : BlockTable (0 .. $bcount);
        .
        .
$endfile
$filename = coverage.adb
        .
        .
```

A.5 Conclusion

This document is primarily meant for the users of gcta. The document has covered all the necessary details of gcta from user's point of view. Regarding syntax of the probe specification file, users will not face any difficulties, because it is exactly same as the syntax for any typical YACC specification file. That is why, the document has not spoken much about the syntax of the probe specification file. The example given in section A.4 shows a typical probe specification file, with the application of the gcta variables/directives.

Appendix B

User's manual for ADACOV

B.1 What is ADACOV?

ADACOV is a Test Coverage Analyzer for Ada95. It takes an ADA program as input and generates an instrumented program, which contains some software probes for coverage monitoring, and coverage reporting operations. The coverage measures, incorporated by ADACOV are :

- Function Coverage
- Branch coverage(including the coverage of Ada Tasks and exceptions)
- Loop coverage
- Call Coverage

A typical coverage report includes the statistics for all kinds of the measures, outlined above, and the following :

- Block-wise coverage statistics
- Cumulative Coverage information for the coverage measures, outlined above, and block-wise coverage statistics.
- A skeleton of the block structure of the program, which can be used for verifying the block-wise coverage statistics.

Options	Meaning
-i indent_width	set no. of blank spaces to be printed for a single indentation (minimum 2). If this option is not specified, the default tab settings will be used for beautifying a program.
-b	Operate in beautify-only mode. If this option is specified, no instrumented program is generated. By default, ADACOV operates as a Test Coverage Analyzer.
-Bm	If this option is specified, the blocks in the output program-file will be marked by using the character '|' and the block number, so as to give a graphical view of the program block-structure.

B.2.2 Output Of ADACOV

Upon successive scanning of the source program, ADACOV generates the instrumented source program with a special ADA package incorporating the Ada functions and procedures for the software probe statements, used in the instrumented program.If ADACOV is operated in "beautify-only" mode, only beautified program will be generated.

If there is any syntax error in the source program, ADACOV does not generate any output; it will of course report the error messages.

B.2.3 Stripping Software Probes

Sometimes user may want to strip the probes inserted in the instrumented program. This situation occurs when user does some modification on the instrumented source file, and wants to remove the probe statements. It can be done by using the following command

stripinstr *instrumented file name* [-o *output filename*]

By default, this command outputs the stripped version of the instrumented program to a file, whose name is same as the instrumented file name, except, it is prefixed by the word "stripped.". However, by giving -o option, user may specify the output filename.

Default Output files

Following are the default output files generated by ADACOV.

adacov_*filename*	Contains the instrumented program. *"filename"* is the name of the input file.
beaut.*filename*	Contains the beautified source program. The line numbers used in the coverage report are with respect to this program. So, if a user wants to cross-reference the coverage output, he/she may use this file. This file is generated automatically, along with the instrumented program file, whenever ADACOV is operated as Test Coverage Analyzer
coverage.ads	Ada package specification for the software monitors, inserted into the instrumented program
coverage.adb	Ada package declaration for coverage.ads.

B.2.4 How to get the coverage statistics

To get the coverage statistics, the instrumented program has to be compiled, using a compiler for Ada95, and then run it. ADACOV has been tested with GNU Ada'95 compiler(gnat3.05)[1] , for DEC Alpha machines, and it has been found working perfectly. The command for compiling the instrumented program, with this compiler, is as follows

[1]Available at ftp://wuarchive.wustl.edu/languages/ada/compiler/gnat/distrib/

gnatmake adacov_*filename*

The next step is to execute the instrumented program. "gnatmake" writes the executable program in the same name as it was provided, without any extension. During the execution of this program, the coverage report will be generated dynamically, and will be written into a file named **Cov.Output**.

Maintaining Coverage History

ADACOV inserts some probe statements for maintaining the coverage history, across different testing sessions. These statements, basically writes the cumulative coverage information in a file named **cov.dat** at the end of a test session, and loads it during a test session, to report the cumulative coverage information. This file stores the following informations

- Test session number
- Coverage records for all the constructs covered so far. This record consists of
  - Type of the construct(e.g. If statement, while statement etc.). This includes block-wise coverage also.
  - Line number at which the construct starts.
  - Line number at which construct ends.

The cumulative coverage reporting, works fine even if the instrumented program is modified, so long as the line numbers for the constructs are not changed. It may give incorrect results, if the modification causes change in starting or ending line numbers of the instrumented program. In that case, user has to delete the file **cov.dat**, and start afresh.

B.3 Troubleshooting

Under normal circumstances, ADACOV works perfectly. However, in some situations, the coverage history file may get corrupted. One such situation occurs, when the program is modified, with some new construct, for which no history was maintained earlier. Another situation could be, deletion of some constructs from the test program. In such cases, the cumulative coverage informations get corrupted. In these circumstances, it is advisable to delete the coverage history file(i.e. file "cov.dat"), and start afresh.

When testing is started for a new program, user must make sure that there is no file named **"cov.dat"** in the test directory. Otherwise, the cumulative coverage information, will get corrupted, or may issue an error message like "Data Error". If this error message is displayed during a test session, the only way to avoid is to delete the history file and start afresh.

B.4 An Example

Here is a sample run of ADACOV. The name of the Ada'95 program file, for which coverage is to be measured, is demo2.adb. To get the cpverage report, user has to give the following command sequence.

```
<73>csealpha3:demo % adacov demo2.adb
No syntax errors detected
To get the coverage information, compile and run adacov_demo2.adb
Beautified program is written into beaut.demo2.adb; You can use this
 file for referencing  coverage information.

<74>csealpha3:demo % gnatmake adacov_demo2
gcc -c adacov_demo2.adb
adacov_demo2.adb:9:11: warning: file name does not match unit name, should be "demo2.adb"
gcc -c coverage.adb
coverage.adb:210:17: warning: "covered" is never assigned a value
gnatbind -x adacov_demo2.ali
gnatlink adacov_demo2.ali
/usr/bin/ld:
Warning: Linking some objects which contain exception information sections
        and some which do not. This may cause fatal runtime exception handling
        problems (last obj encountered without exceptions was
    /2d1/ska/dosins/gnat.dec/lib/gcc-lib/alpha-dec-osf3.2/2.7.2/adalib/s-unstyp.o).

<75>csealpha3:demo % adacov_demo2
```

Here is a sample coverage report, generated after executing the last command, given above.

```
------------------------------------
- Current test session no =   2          Program File : demo2.adb
------------------------------------
 Line numbers are with reference to the file beaut.demo2.adb

Branch Coverage Information
-------------------------------
Statement            Covered          Total Nos.
-------------------------------------------------
 If_Statement            0               3
 While_Statement         1               1
        From line no.   15 to  18 (  21 times )
 Simple_loop             1               1
        From line no.   50 to  71 (   3 times )
 Exception_When_stmt     0               1
----------------------------------------------
Total                    2               6


Function/Procedure Call  Coverage Information
----------------------------------------------


No. of Procedure/Function Call       Covered
---------------------------------------------
 22 ( Procedure  Call           )     18
        At line no.  32 (   1 times )
        At line no.  33 (   1 times )
        At line no.  34 (   1 times )
        At line no.  35 (   1 times )
        At line no.  36 (   1 times )
        At line no.  37 (   1 times )
        At line no.  38 (   1 times )
        At line no.  47 (   1 times )
        At line no.  48 (   1 times )
        At line no.  49 (   1 times )
        At line no.  16 (  21 times )
        At line no.  19 (   3 times )
        At line no.  51 (   3 times )
        At line no.  61 (   3 times )
        At line no.  62 (   3 times )
        At line no.  65 (   2 times )
        At line no.  66 (   2 times )
        At line no.  67 (   2 times )
---------------------------------------------
 22                                   18
```

```
Return/Goto  Coverage Information
------------------------------------

No. of Return/goto Statement         Covered
---------------------------------------------
---------------------------------------------
  0                                    0


Blockwise Coverage Information
--------------------------

Total Blocks :   8
Blocks Covered :
  0  (From line no  14 to  20 :   3 Times )
  1  (From line no  15 to  18 :  21 Times )
  2  (From line no  23 to  75 :   1 Times )
  3  (From line no  50 to  71 :   3 Times )

Total Number of Blocks Covered :   4
Blocks Not Covered So far:
  4,     5,     6,     7,


Cumulative Coverage Information :
---------------------------------
Total No. of testing session :  2


Branch Coverage Information
-------------------------------
Statement              Covered          Total Nos.
-------------------------------------------------
 If_Statement              0              3
 While_Statement           1              1
         From line no.  15 to  18 (  42 times )
 Simple_loop               1.             1
         From line no.  50 to  71 (   6 times )
 Exception_When_stmt       0              1
----------------------------------------------
Total                      2              6


Function/Procedure Call  Coverage Information
----------------------------------------------
```

```
No. of Procedure/Function Call        Covered
-----------------------------------------------
 22 ( Procedure  Call              )     18
         At line no.  32 (   2 times )
         At line no.  33 (   2 times )
         At line no.  34 (   2 times )
         At line no.  35 (   2 times )
         At line no.  36 (   2 times )
         At line no.  37 (   2 times )
         At line no.  38 (   2 times )
         At line no.  47 (   2 times )
         At line no.  48 (   2 times )
         At line no.  49 (   2 times )
         At line no.  16 (  42 times )
         At line no.  19 (   6 times )
         At line no.  51 (   6 times )
         At line no.  61 (   6 times )
         At line no.  62 (   6 times )
         At line no.  65 (   4 times )
         At line no.  66 (   4 times )
         At line no.  67 (   4 times )
-----------------------------------------------
 22                                      18


Return/Goto  Coverage Information
--------------------------------------

No. of Return/goto Statement          Covered
-----------------------------------------------
-----------------------------------------------
  0                                        0

Blockwise Coverage Information
---------------------------

Total Blocks :   8
Blocks Covered :
  0  (From line no  14 to  20 :   6 Times )
  1  (From line no  15 to  18 :  42 Times )
  2  (From line no  23 to  75 :   2 Times )
  3  (From line no  50 to  71 :   6 Times )

Total Number of Blocks Covered :   4
Blocks Not Covered So far:
  4,     5,     6,     7,

Block Structure for the program demo2.adb is  shown below :
Only executable blocks are shown; Unnumbered blocks are declarations
```

```
To generate the beautified program with the blocks shown graphically, use the command
         adacov -b -Bm [other options] demo2.adb

|       --s14
|       |       --s15
|       |       |
|       B0      B1
|       |       |
|       |       --e18
|       --e20
--s23
|       --s50
|       |       --s52
|       |       |
|       |       B4
|       |       |
|       |       --e54
|       |       --s55
|       |       |
|       B3      B5
|       |       |
|       |       --e57
|       |       --s58
B2      |       |
|       |       B6
|       |       |
|       |       --e60
|       --e71
|       --s73
|       |
|       B7
|       |
|       --e74
|
--e75
```

In the skeleton block structure of the input program, which is shown at the end of the coverage report, the starting and ending line numbers of a block are indicated by the prefixes ‘s’ and ‘e’ respectively.

B.5 Conclusion

This document has given all the necessary information about the usage of ADACOV. The detailed command synopsis, and the various features for ADACOV have been outlined in this document.